Assembly Programming IV

CSE 351 Spring 2017

Instructor:

Ruth Anderson

Teaching Assistants:

Dylan Johnson

Kevin Bi

Linxing Preston Jiang

Cody Ohlsen

Yufang Sun

Joshua Curtis

Administrivia

- Homework 2 due this Wednesday (4/19)
- Lab 2 (x86-64) due next Wednesday (4/26)
    - Learn to read x86-64 assembly and use GDB

Review

- 3 ways to set condition codes are:

  cmp, test, arithmetic instructions

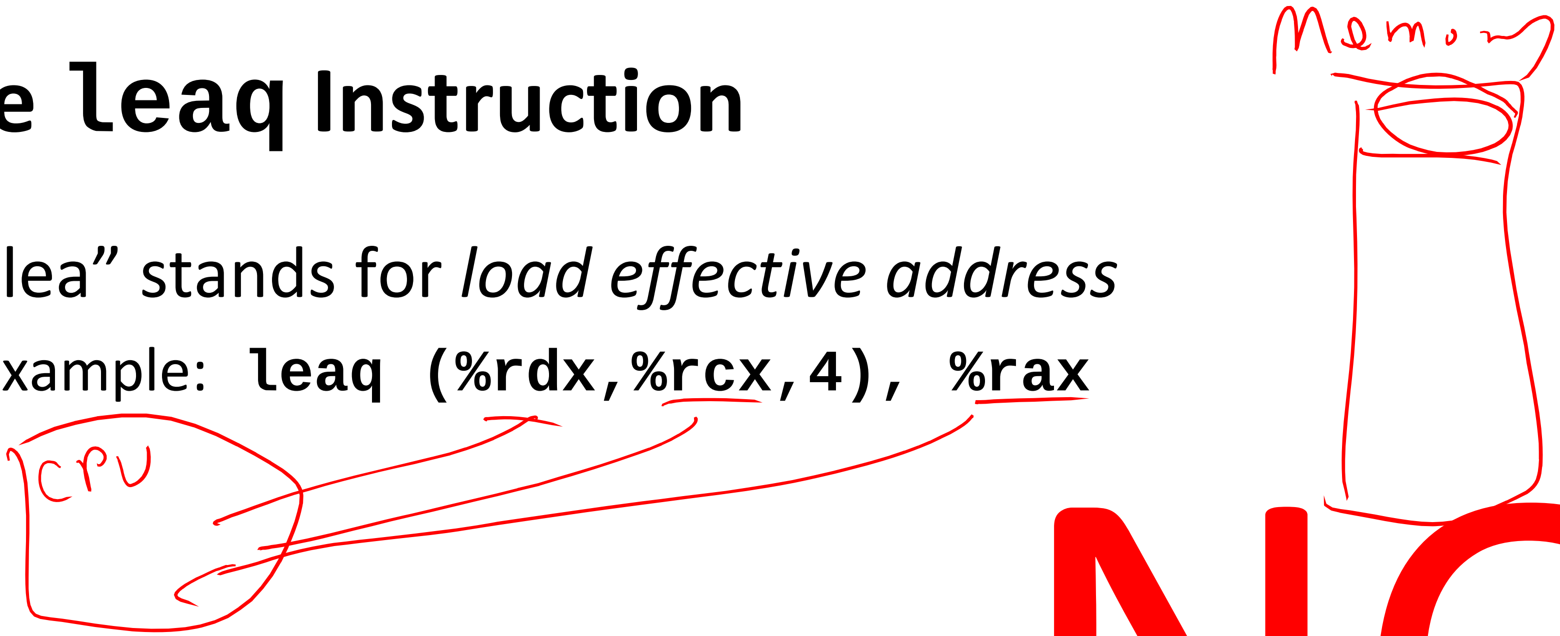
- 2 ways to use condition code are:

  jmp, set

- Does leaq set condition codes? no

The `leaq` Instruction

- “lea” stands for *load effective address*
  - Example: **`leaq (%rdx,%rcx,4), %rax`**



Does the `leaq` instruction go to memory?

NO

“`leaq` – it just does math”

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- **Loops**
- Switches

Expressing with Goto Code

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

conditional jump

cmp

jle

unconditional jump → goto Done;

jmp

labels (addresses)

- C allows `goto` as means of transferring control (`jump`)
    - Closer to assembly programming style
    - Generally considered bad coding style
    - This is just to help you understand assembly code generated by the compiler. Do NOT use `goto` in your C code!

Compiling Loops

C/Java code:

Test

```
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:

```
loopTop:   testq  %rax, %rax
           je     loopDone
           <loop body code>
           jmp    loopTop
loopDone:
```

~Test

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time
- Most important to consider:
  - When should conditionals be evaluated? (*while* vs. *do-while*)
  - How much jumping is involved?

Compiling Loops

C/Java code:

```
while ( Test ) {
    Body
}
```

Goto version

```
Loop: if ( !Test )  goto Exit;
      Body
      goto Loop;
Exit:
```

- What are the Goto versions of the following?
  - Do…while: Test and Body
  - For loop: Init, Test, Update, and Body

```
do {
    Body
} while (Test);
```

```
for (Init; Test; Update) {
    Body
}
```

Do…while

```
Loop:
    Body
    if ( Test )  goto Loop;
```

For Loop

```
        Init
Loop:   if (~Test)  goto Exit;
        Body
        Update
        goto Loop;
Exit:
```

Compiling Loops

all jump instructions update the program counter (%rip)

While Loop:

C:

```
while ( sum != 0 ) {
    <loop body>
}
```

Test

x86-64:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

sum == 0

~Test

Do-while Loop:

C:

```
do {
    <loop body>
} while ( sum != 0 )
```

Test

x86-64:

```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```

Test

While Loop (ver. 2):

C:

```
while ( sum != 0 ) {
    <loop body>
}
```

Test

x86-64:

```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```

~Test

do-while loop

Test

For Loop → While Loop

For Version

```
for (Init; Test; Update)

        Body
```

While Version

```
Init;

while (Test) {

        Body

        Update;

}
```

Caveat: C and Java have `break` *and* `continue`

- *Conversion works fine for* `break`
    - *Jump to same label as loop exit condition*
- *But not* `continue`*: would skip doing Update, which it should do with for-loops*
    - *Introduce new label at Update*

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- **Switches**

```
long switch_ex
   (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
    - Here: 5 & 6
- Fall through cases
    - Here: 2
- Missing cases
    - Here: 4

- How to implement this?

Jump Tables

- Compiles sometimes Implement switch statements with:
  - Jump table
  - Uses the Indirect jump instruction
- Why? When?

Jump Table Structure

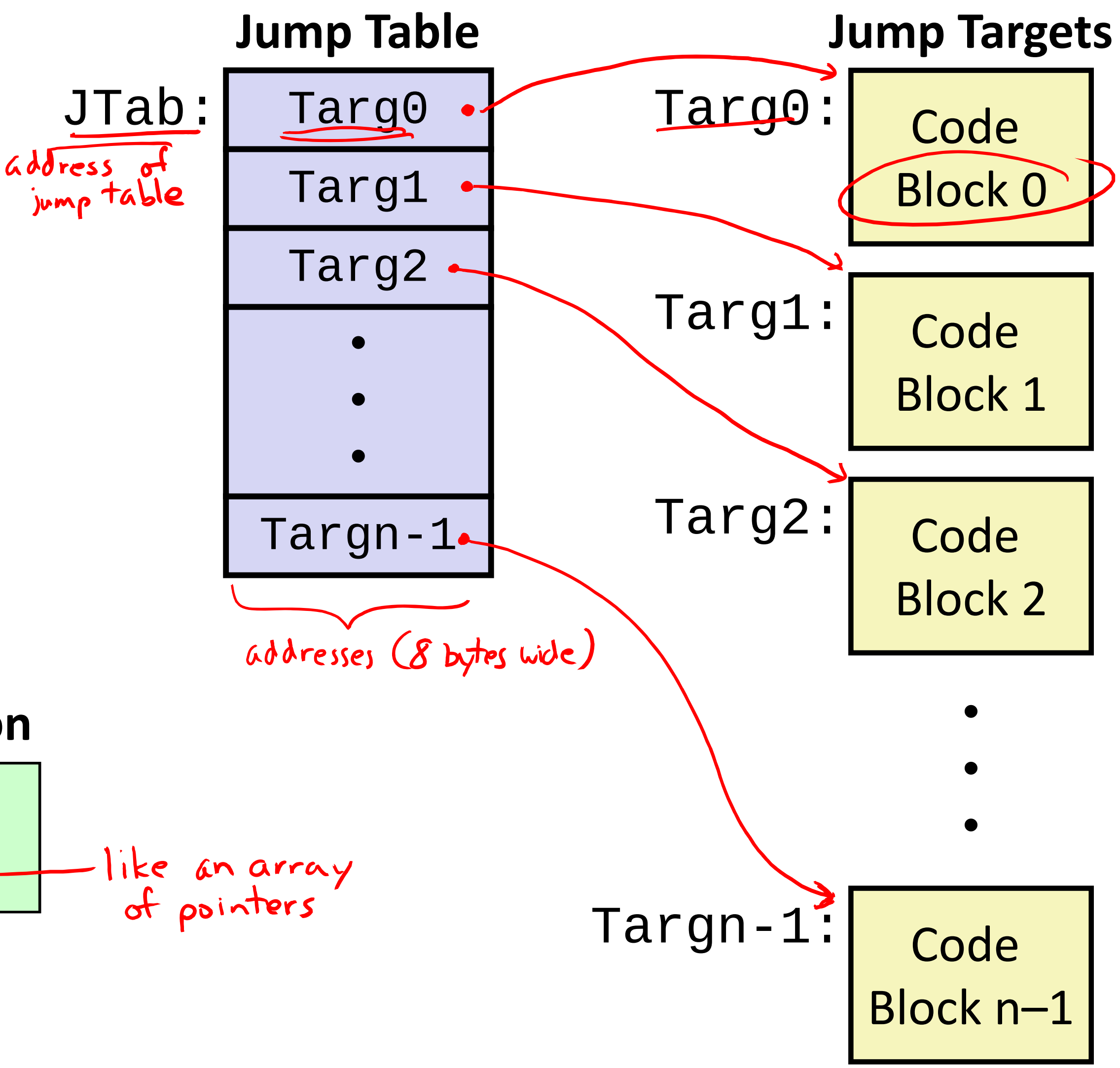
Switch Form

```
switch (x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Approximate Translation

```
target = JTab[x];
goto target;
```

like an array of pointers



Jump Table

Jump Targets

JTab:

address of jump table

Targ0

Targ1

Targ2

•
•
•

Targn-1

addresses (8 bytes wide)

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targn-1: Code Block n–1

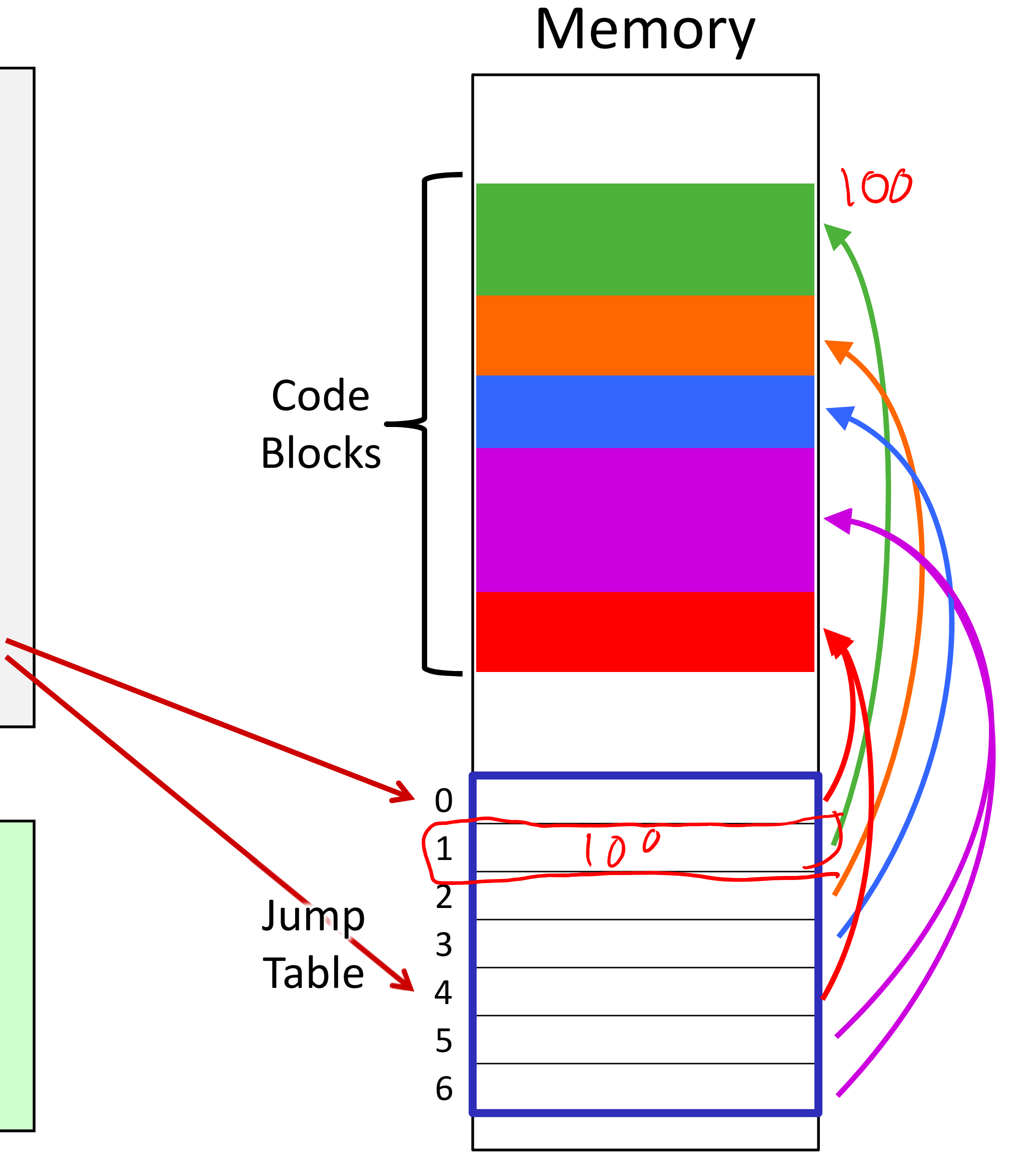
Jump Table Structure

C code:

```
switch (x) {
   case 1: <some code>
           break;
   case 2: <some code>
   case 3: <some code>
           break;
   case 5:
   case 6: <some code>
           break;
   default: <some code>
}
```

Use the jump table when x ≤ 6:

```
if (x <= 6)
   target = JTab[x];
   goto target;
else
   goto default;
```



Switch Statement Example

Register	Use(s)
`%rdi`	1st argument (x)
`%rsi`	2nd argument (y)
`%rdx`	3rd argument (z)
`%rax`	Return value

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
       . . .
    }
    return w;
}
```

where?

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi         # x:6
    ja      .L8              # default
    jmp     *.L4(,%rdi,8)    # jump table
```

a b

jump to default case if x>6 (unsigned)

Note: compiler chose to not initialize `w`

Take a look!

https://godbolt.org/g/DnOmXb

jump above – unsigned > catches negative default cases

-1 > 6U → jump to default case

Switch Statement Example

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section   .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

address

following data is a "quad word" = 8 bytes

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # default
    jmp     *.L4(,%rdi,8)     # jump table
```

Indirect jump

D + Ri * S

addr of jump table

x

sizeof (void*)

dereference Mem operator and store that in %rip

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes (address)
  - Base address at `.L4`

- **Direct jump:** `jmp .L8`
  - Jump target is denoted by label `.L8`

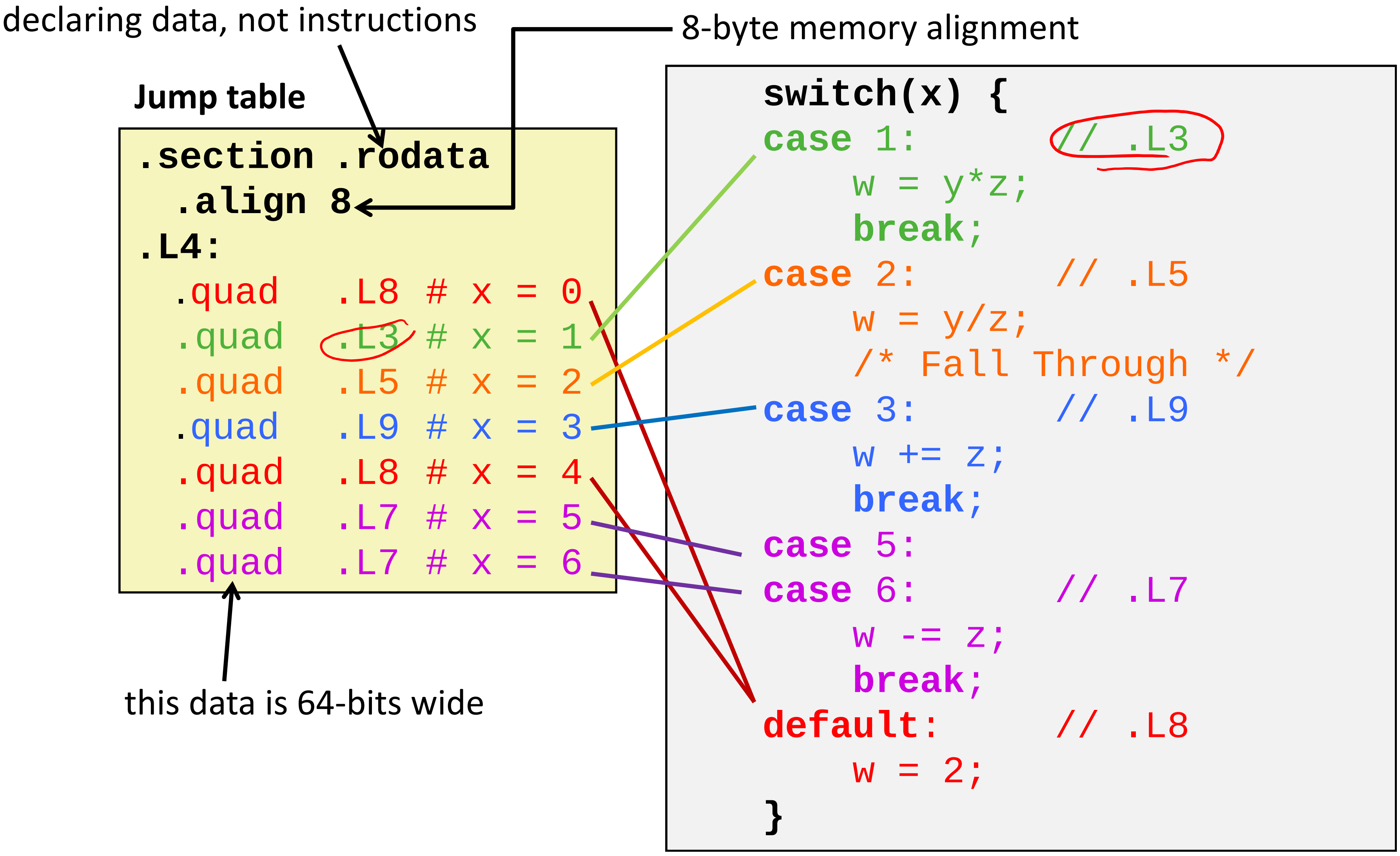
- **Indirect jump:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective address `.L4 + x*8`
    - Only for $0 \le x \le 6$

Mem[D+ Reg[Ri]*S]

Jump table

```
.section   .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Jump Table



declaring data, not instructions

8-byte memory alignment

Jump table

```
.section .rodata
  .align 8
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

this data is 64-bits wide

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

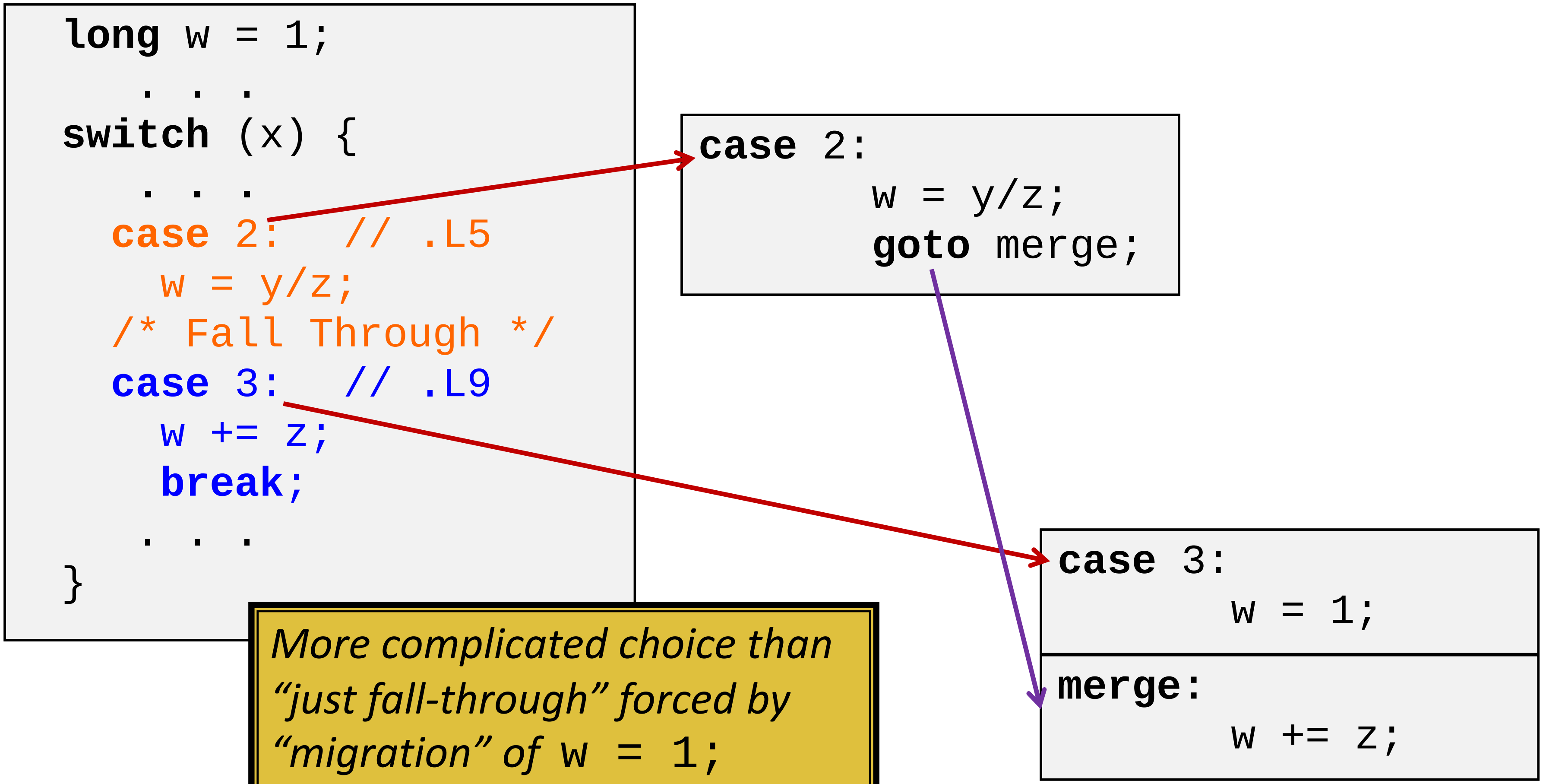
Code Blocks (x == 1)

Register	Use(s)
`%rdi`	1st argument (x)
`%rsi`	2nd argument (y)
`%rdx`	3rd argument (z)
`%rax`	Return value

```
switch(x) {
  case 1:     // .L3
    w = y*z;
    break;
   . . .
}
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

Handling Fall-Through



```
long w = 1;
   . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
case 2:
       w = y/z;
       goto merge;
```

```
case 3:
       w = 1;
```

```
merge:
       w += z;
```

More complicated choice than “just fall-through” forced by “migration” of `w = 1;`

- *Example compilation trade-off*

Code Blocks (x == 2, x == 3)

Register	Use(s)
`%rdi`	1st argument (x)
`%rsi`	2nd argument (y)
`%rdx`	3rd argument (z)
`%rax`	Return value

```
long w = 1;
   . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
.L5:                           # Case 2:
   movq     %rsi, %rax  #  y in rax
   cqto                        #  Div prep
   idivq    %rcx              #  y/z
   jmp      .L6                #  goto merge
.L9:                           # Case 3:
   movl     $1, %eax    #  w = 1
.L6:                           # merge:
   addq     %rcx, %rax  #  w += z
   ret
```

Code Blocks (rest)

Register	Use(s)
`%rdi`	1st argument (x)
`%rsi`	2nd argument (y)
`%rdx`	3rd argument (z)
`%rax`	Return value

```
switch (x) {
   . . .
  case 5:  // .L7
  case 6:  // .L7
    w -= z;
    break;
  default: // .L8
    w = 2;
}
```

```
.L7:                    # Case 5,6:
  movl  $1, %eax       #  w = 1
  subq  %rdx, %rax     #  w -= z
  ret
.L8:                    # Default:
  movl  $2, %eax       #  2
  ret
```

Roadmap

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

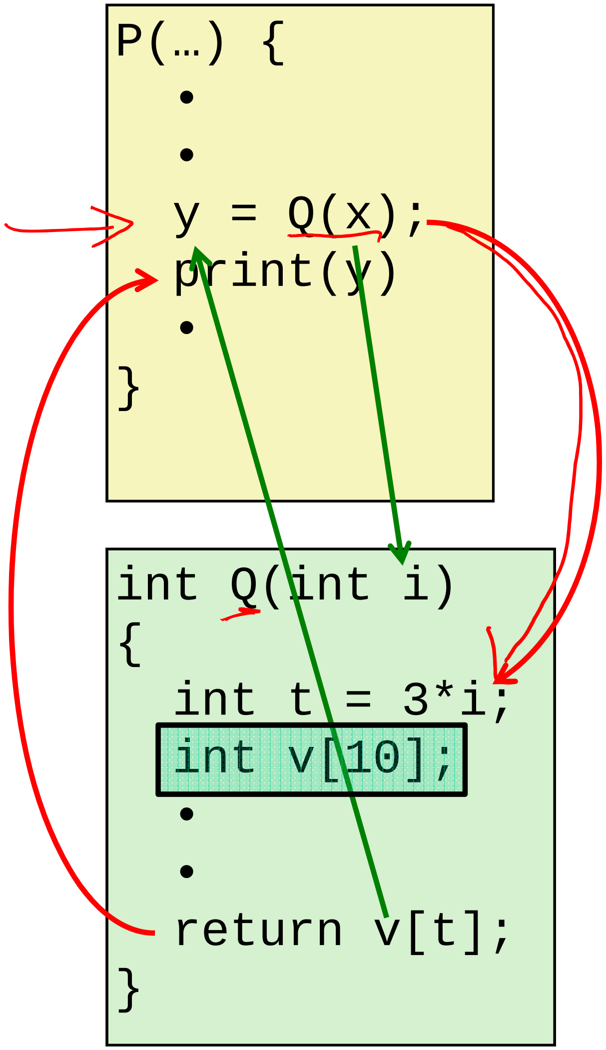
OS:

Computer system:

Mechanisms required for *procedures*

1) Passing control
   - To beginning of procedure code
   - Back to return point
2) Passing data
   - Procedure arguments
   - Return value
3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return

- All implemented with machine instructions!
  - An x86-64 procedure uses only those mechanisms required for that procedure



```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Questions to answer about Procedures

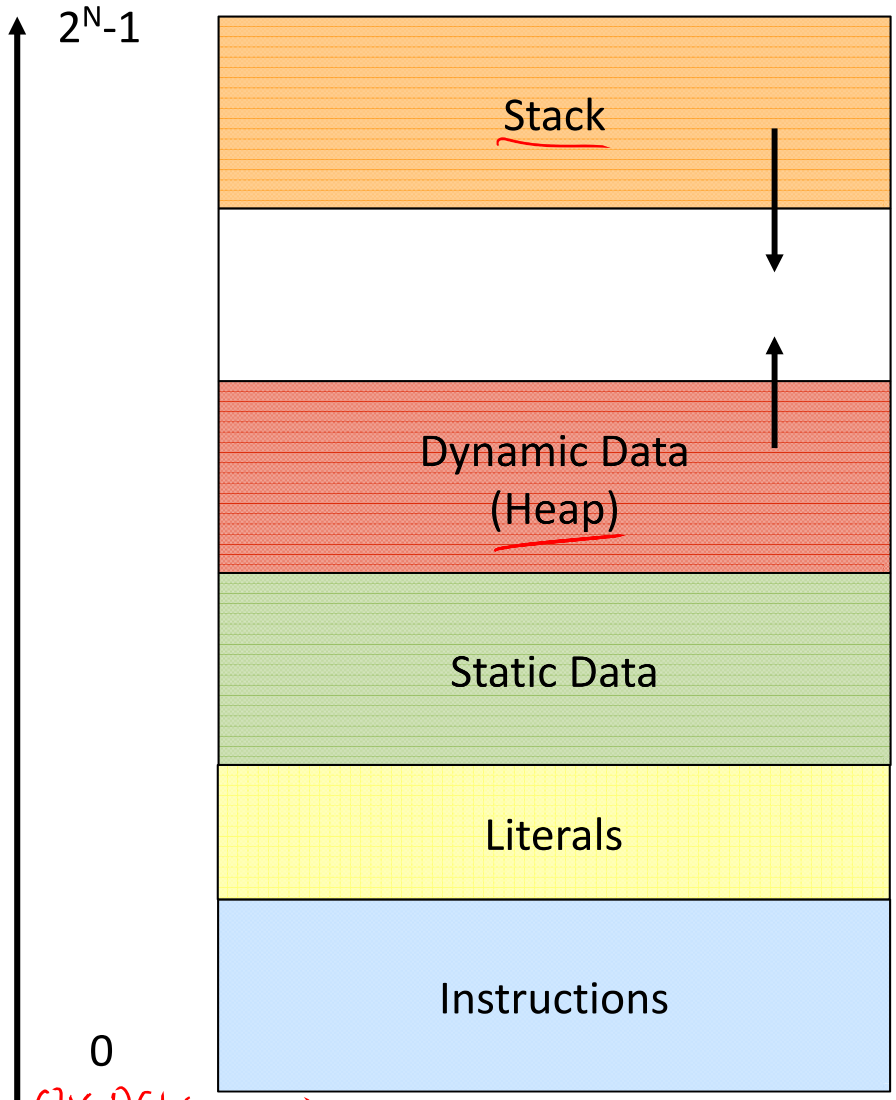
- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return?

- To answer some of these questions, we need a *call stack* …

Procedures

- **Stack Structure**
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Memory Layout



0xFFF...F

High Addresses

2^N-1

Stack

local variables; procedure context

Dynamic Data (Heap)

variables allocated with *new* or *malloc*

Memory Addresses

Static Data

static variables (including global variables (C))

Literals

large constants (*e.g.* "example")

Instructions

program code

Low Addresses

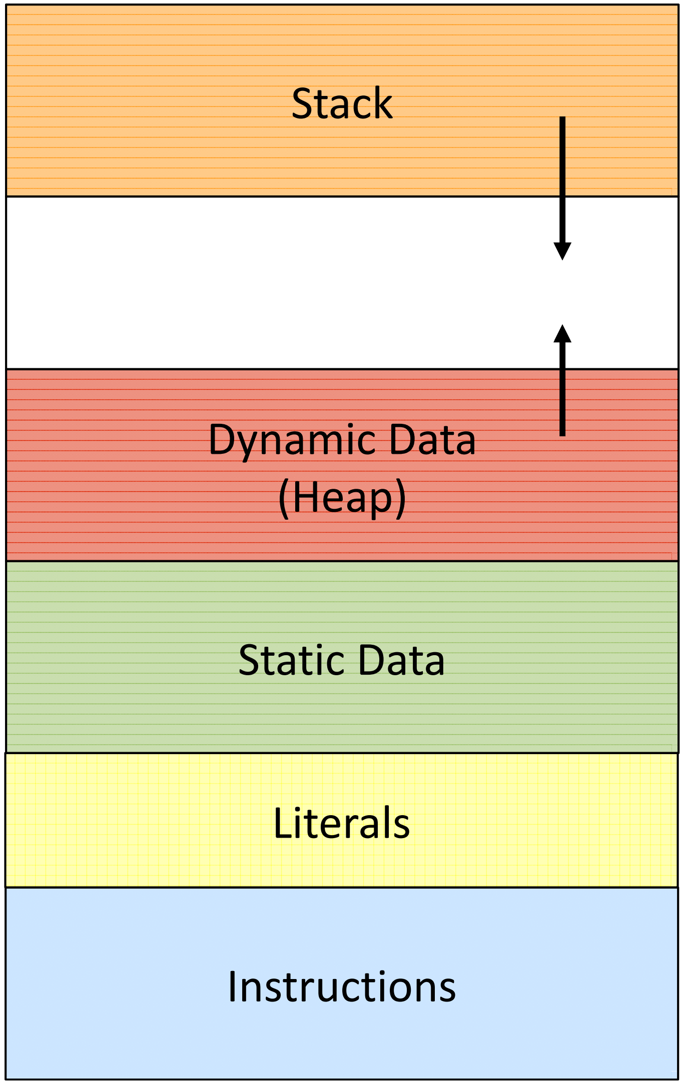
0

0x000..00

Memory Permissions

segmentation faults?

accessing memory in a way that you are not allowed to



writable; not executable	Stack	Managed "automatically" (by compiler)
		grow towards each other to maximize use of space
writable; not executable	Dynamic Data (Heap)	Managed by programmer
writable; not executable	Static Data	Initialized when process starts
read-only; not executable	Literals	Initialized when process starts
read-only; executable	Instructions	Initialized when process starts

x86-64 Stack

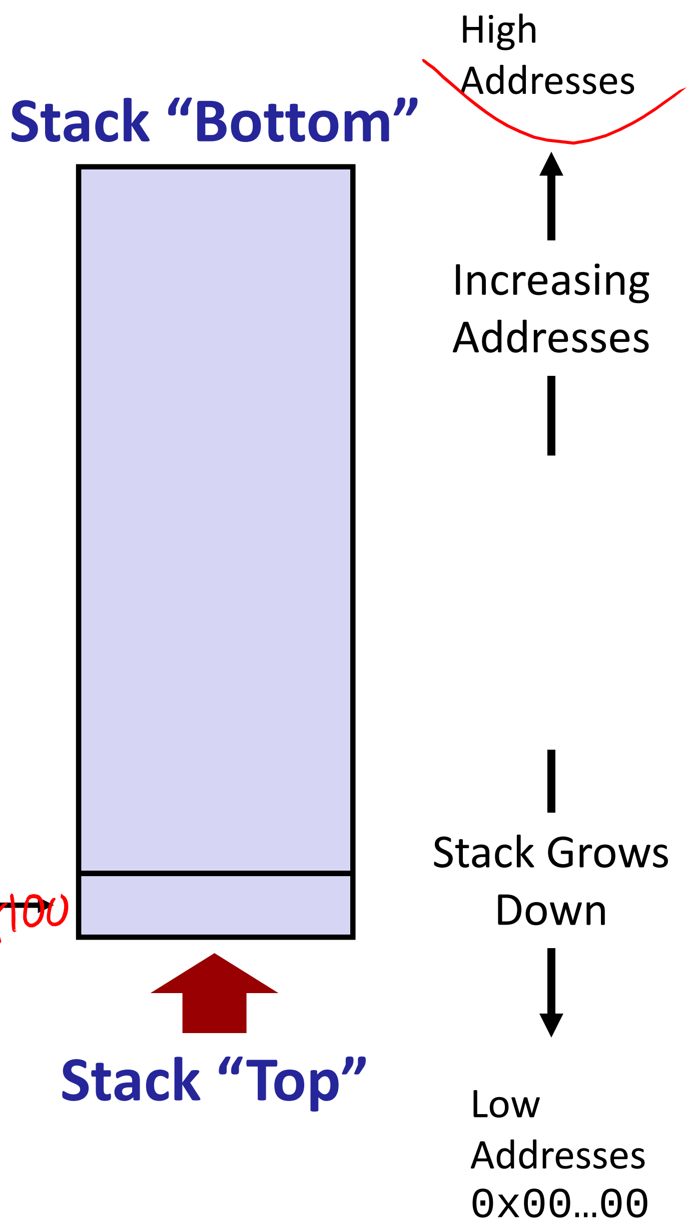
- Region of memory managed with stack "discipline"
  - Grows toward lower addresses
  - Customarily shown "upside-down"

- Register `%rsp` contains *lowest* stack address
  - `%rsp` = address of *top* element, the most-recently-pushed item that is not-yet-popped

Stack Pointer: `%rsp` ~~0x100~~

CPU

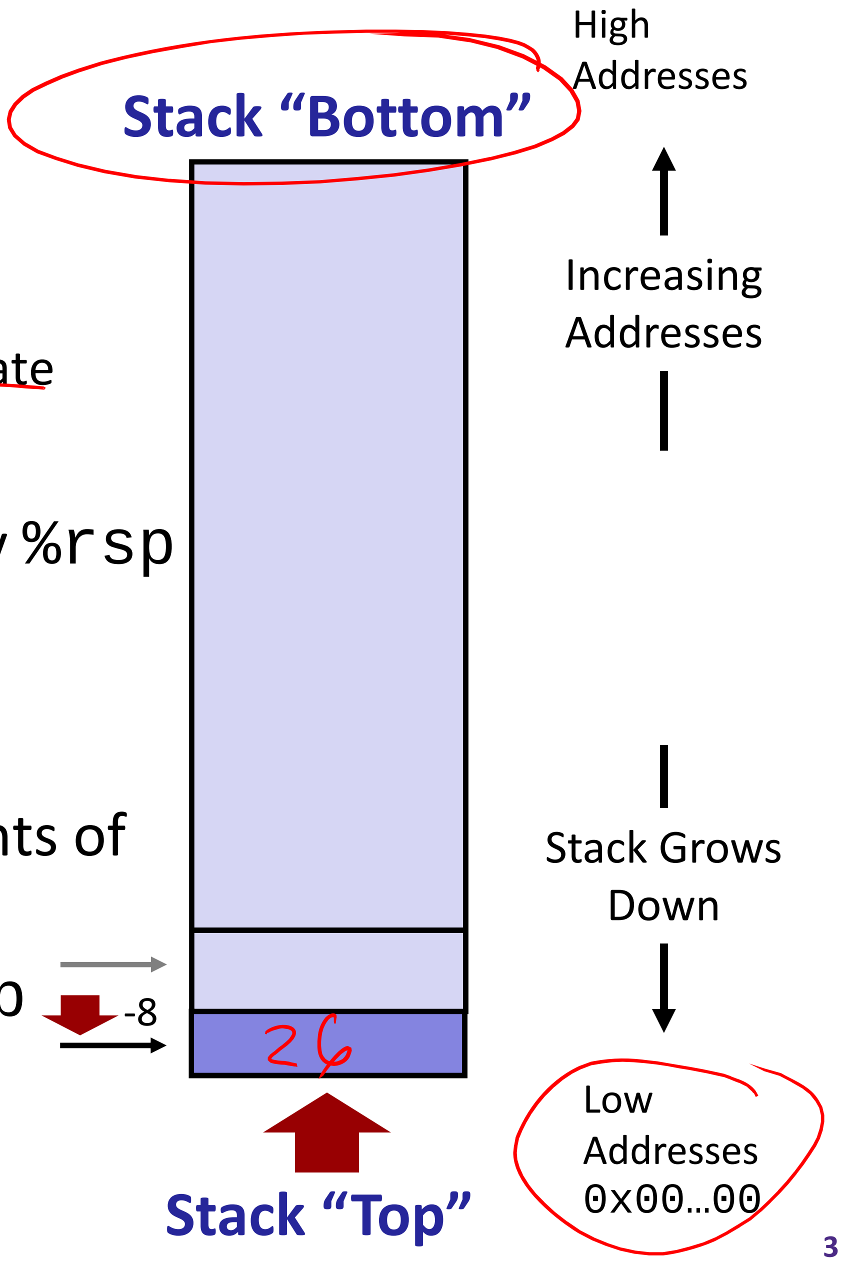
rsp ~~0x100~~



x86-64 Stack: Push

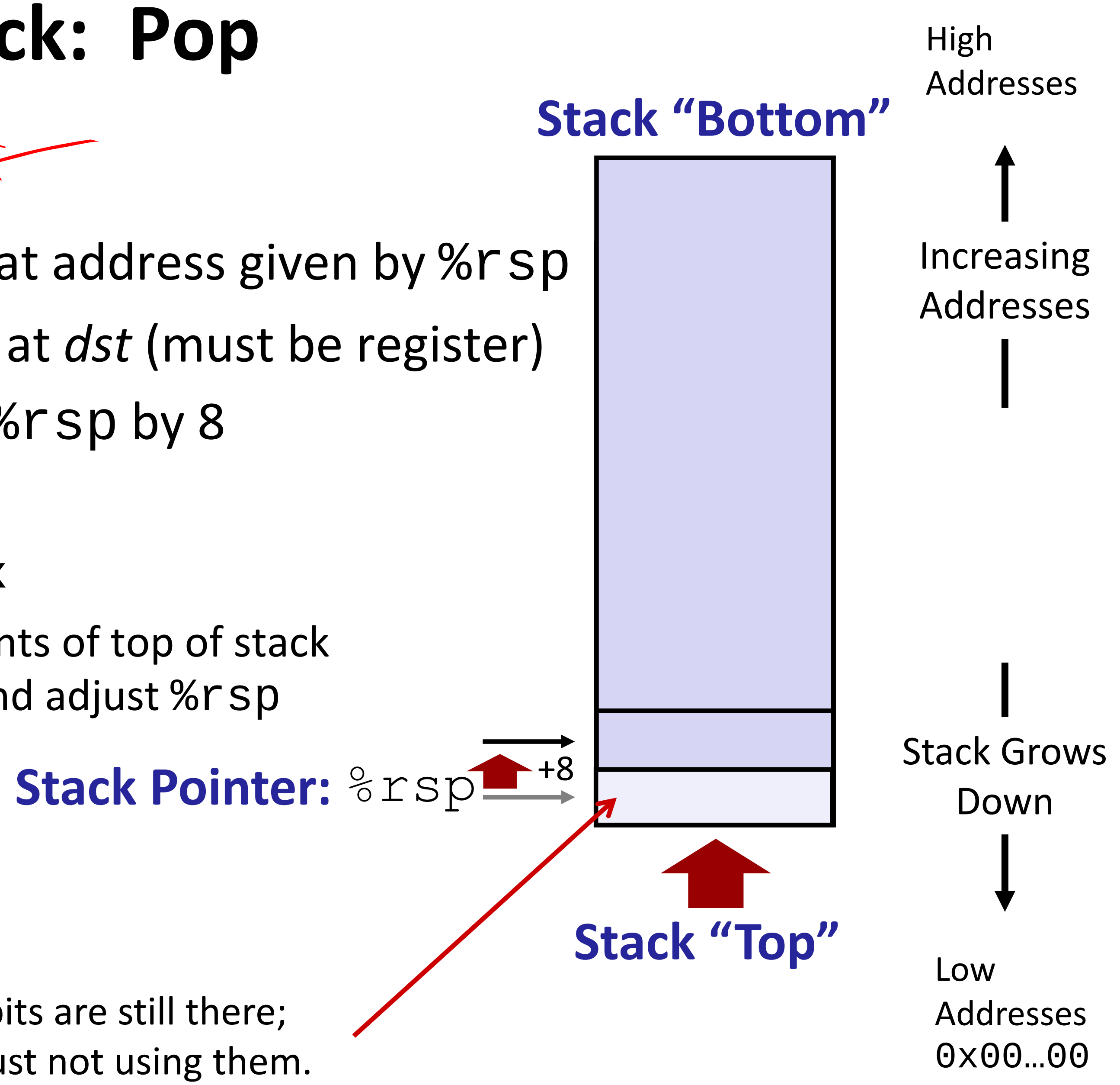
- `pushq` *src* 26
  - ⓪ Fetch operand at *src*
    - *Src* can be reg, memory, immediate
  - ① ***Decrement*** `%rsp` by 8
  - ② Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack

Stack Pointer: `%rsp`



x86-64 Stack: Pop

- popq *dst*
  - ⓪ Load value at address given by `%rsp`
  - ① Store value at *dst* (must be register)
  - ② ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

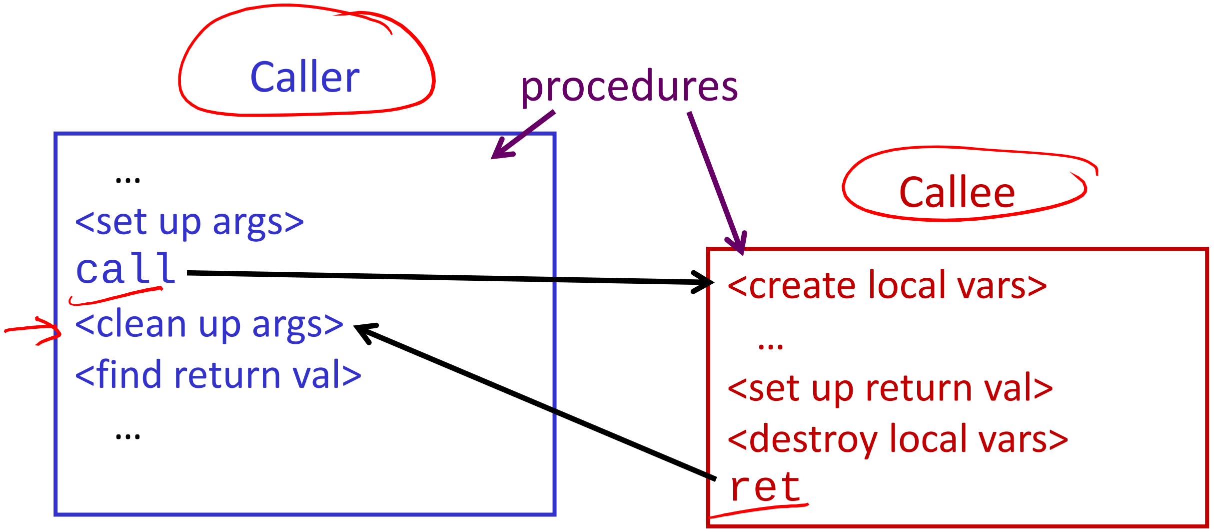


Those bits are still there;
we're just not using them.

Procedures

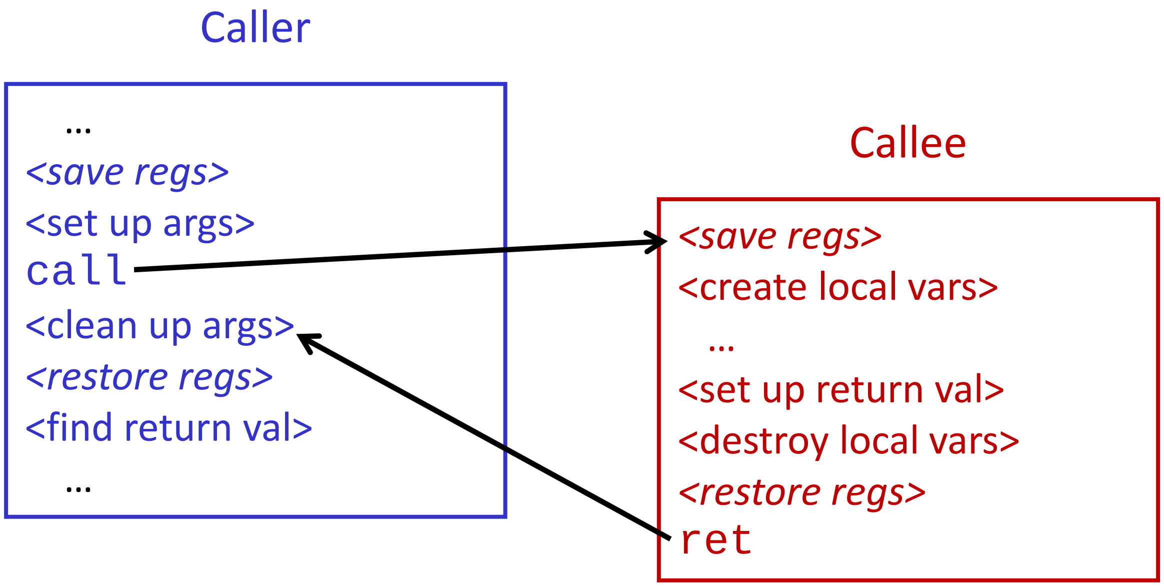
- Stack Structure
- **Calling Conventions**
  - **Passing control**
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Call Overview



- **Callee** must know where to find args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (*e.g.* no arguments)

Procedure Call Overview



Caller

…
<save regs>
<set up args>
`call`
<clean up args>
<restore regs>
<find return val>
…

Callee

<save regs>
<create local vars>
…
<set up return val>
<destroy local vars>
<restore regs>
`ret`

- The *convention* of where to leave/find things is called the calling convention (or procedure call linkage)
    - Details vary between systems
    - We will see the convention for x86-64/Linux in detail
    - What could happen if our program didn't follow these conventions?

Code Examples

caller

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

Compiler Explorer:
https://godbolt.org/g/52Sqxj

executable disassembly

```
0000000000400540 <multstore>:
  400540: push   %rbx             # Save %rbx
  400541: movq   %rdx,%rbx        # Save dest
  400544: call   400550 <mult2>   # mult2(x,y)
  400549: movq   %rax,(%rbx)      # Save at dest
  40054c: pop    %rbx             # Restore %rbx
  40054d: ret                     # Return
```

these are instruction addresses

callee

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax   # a
  400553:  imulq  %rsi,%rax   # a * b
  400557:  ret                # Return
```

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label`
  1) Push return address on stack (*why? which address?*)
  2) Jump to ***`label`***

① move %rsp down
② store ret addr at %rsp
③ label → %rip

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label`
  1) ~~Push~~ return address on stack (*why? which address?*)
  2) Jump to ***`label`***

① move %rsp down
② store ret addr at %rsp
③ label → %rip

- Return address:
  - Address of instruction immediately after **`call`** instruction
  - Example from disassembly:

```
400544: call   400550 <mult2>
400549: movq   %rax,(%rbx)
```

Return address = **`0x400549`**

next instruction happens to be a move, but could be anything

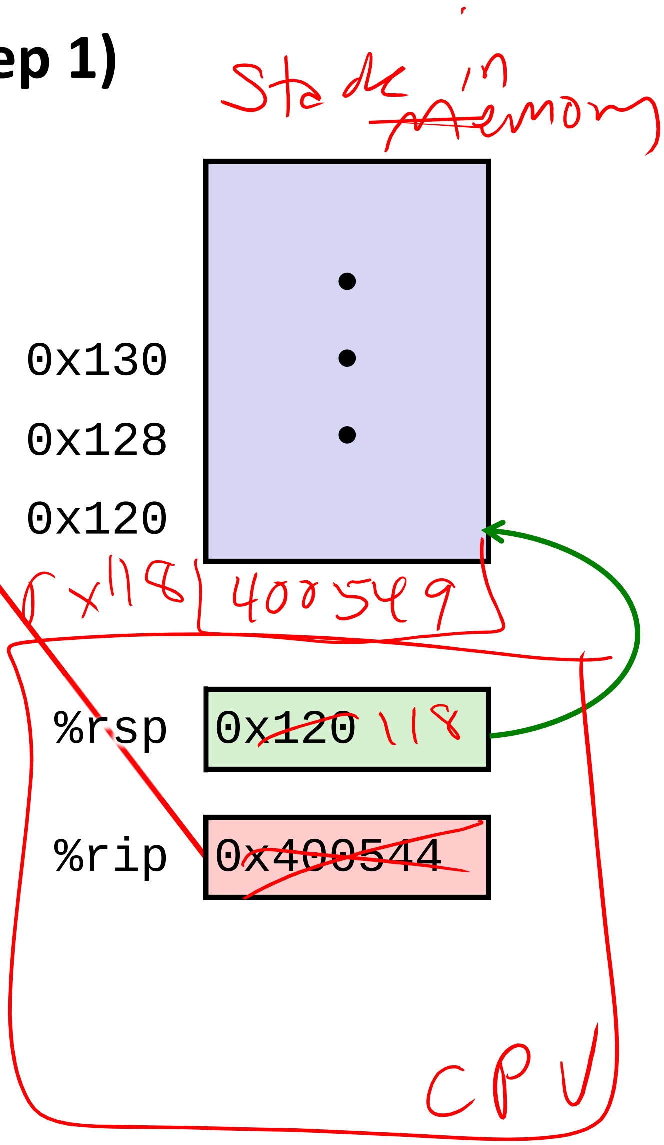
- Procedure return: `ret`
  1) Pop return address from stack
  2) Jump to address

① read ret addr at %rsp (into %rip)
② move %rsp up

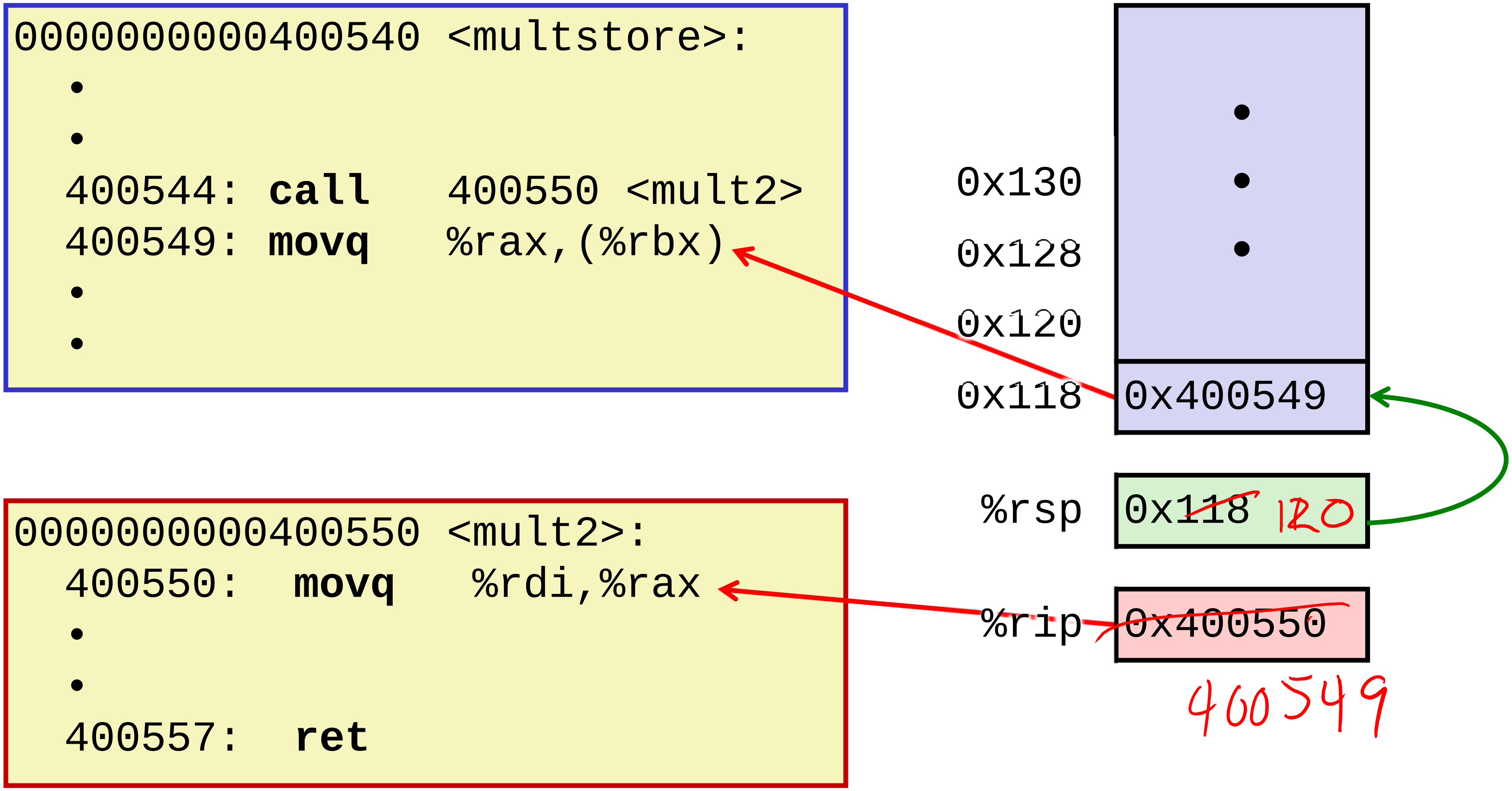
Procedure Call Example (step 1)

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



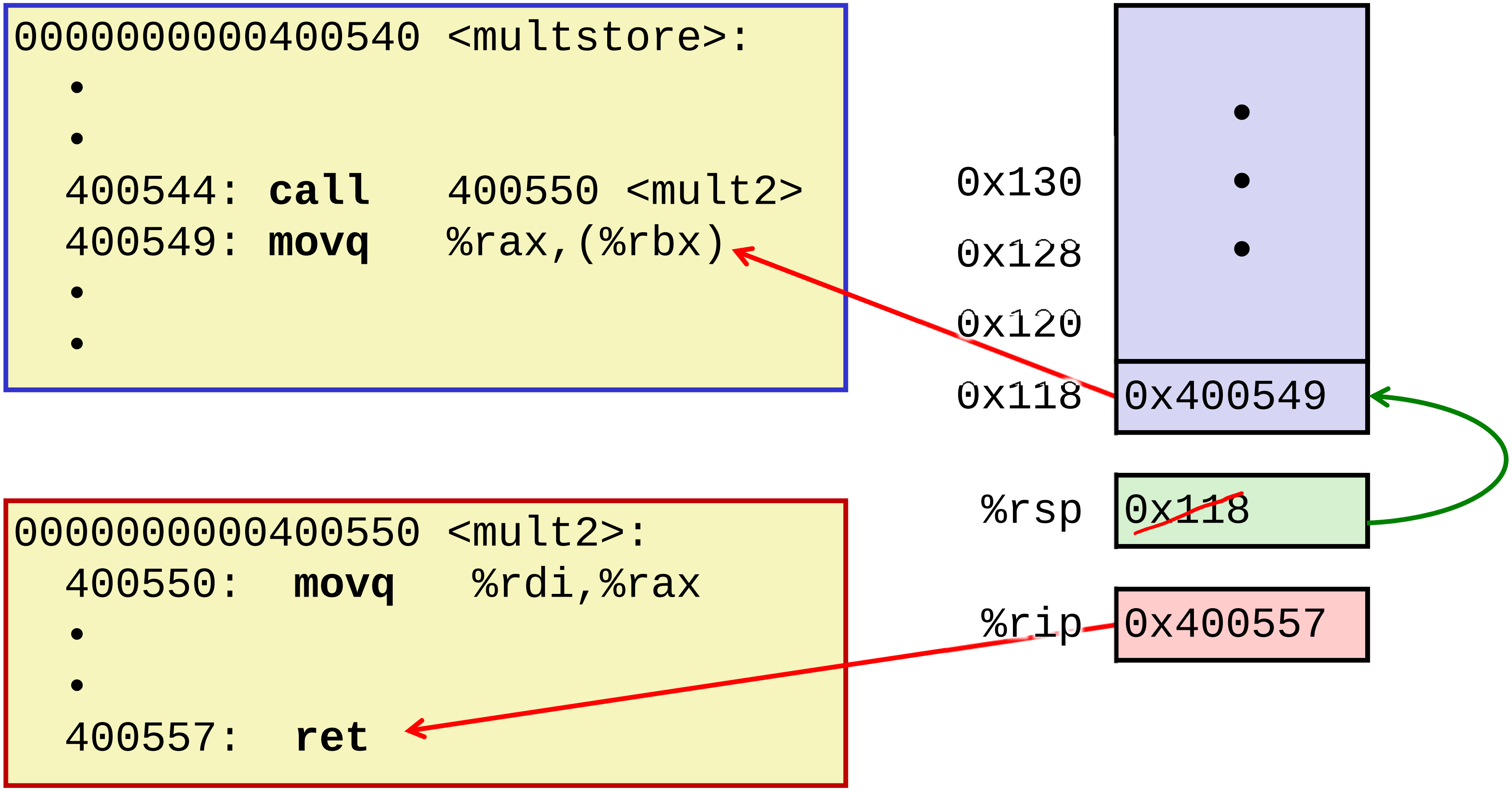
Procedure <u>Call</u> Example (step 2)



```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```

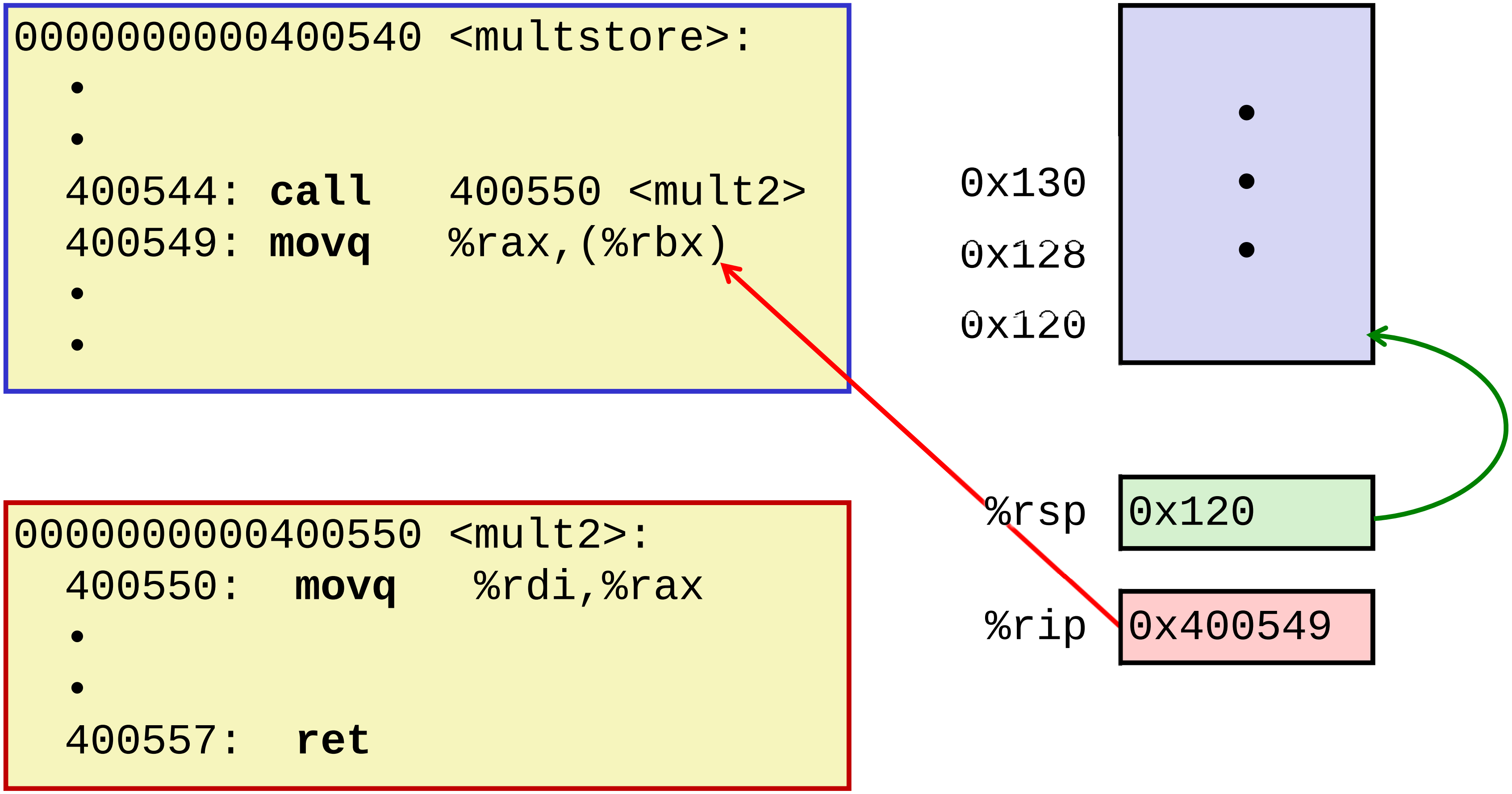
Procedure Return Example (step 1)



```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```

Procedure Return Example (step 2)



```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```